Remarks

Applicants respectfully request reconsideration of the application in view of the foregoing amendments and following remarks.

Claims 1 and 4-42 are currently pending in the Application. Claims 1, 6, 9, 10, 13, 16, 18, 20, and 21 are independent. The Office action of January 26, 2005 ["Action"] rejects claim 21 under 35 U.S.C. § 102(a) as being anticipated by "Attribute Programming with Visual C++" by Richard Grimes ["Grimes"]. Claims 1, 4-20, and 22-36 are rejected under 35 U.S.C. § 103(a) as being unpatentable over Grimes in view of "Compiler Principles, Techniques, and Tools" by Aho et al. ["Aho"]. Additionally, claims 1, 4-10, 13-20, 22, 24-25, 27-30, 32-35 are rejected under 35 U.S.C. 103(a) as being unpatentable over Aho in view of U.S. Patent No. 5,467,472 to Williams ["Williams"]. Applicants respectfully disagree.

I. Claims 1, 4-20, and 22-36 Are Allowable over Grimes in View of Aho

Claims 1, 4-19, and 22-36

Claims 1, 4-19, and 22-36 have been rejected under 35 U.S.C. § 103(a) as being unpatentable over Grimes in view of Aho. Independent claims 1, 6, 9, 10, 13, 16, and 18 have been amended in order to clarify certain aspects. No new matter has been added.

Grimes and Aho, taken separately or in combination, fail to teach or suggest at least one limitation from each of independent claims 1, 6, 9, 10, 13, 16, and 18 as amended. Thus, the references fail to teach or suggest at least one limitation from each of claims 1, 4-19, and 22-36, and these claims are allowable.

Claim 1, as amended, recites:

> a converter module that converts the plural tokens into an intermediate representation, wherein the intermediate representation includes a symbol table and a tree that unifies representation of the programming language code and the embedded definition language information, *wherein at least some of the embedded definition language information is represented in the tree without creating new programming language code for the at least some of the embedded definition language information....*

Claim 6, as amended, recites:

> a converter module that converts the plural tokens into an intermediate representation comprising a symbol table and a parse tree, wherein:...

at least some of the interface definition language constructs are represented in the parse tree without creating new programming language constructs for the at least some of the interface definition language constructs.

Claim 9, as amended, recites:

A computer readable medium having stored thereon a data structure representing a unified interface definition language and programming language parse tree for a file having a combination of programming language code and embedded interface definition language information,...

wherein at least some of the embedded definition language information is represented in the parse tree without creating new programming language code for the at least some of the embedded definition language information.

Claim 10, as amended, recites:

converting the one or more input files into one or more output code files that include fragments of definition language information, ... *wherein at least some of the definition language constructs are represented in the tree without creating new programming language constructs for the at least some of the definition language constructs....*

Claim 13, as amended, recites:

converting the plural tokens into an intermediate representation, wherein the converting comprises building a tree that unifies representation of the programming language code and the embedded definition language information and *the building comprises representing at least some of the embedded definition language information in the tree without creating new programming language code for the at least some of the embedded definition language information ...*

Claim 16, as amended, recites:

building a tree that unifies representation of the embedded definition language information and the programming language code, *wherein the building comprises representing at least some of the embedded definition language information in the tree without creating new programming language code for the at least some of the embedded definition language information.*

Claim 18, as amended, recites:

a converter module that converts the plural programming language tokens and plural definition language tokens into an intermediate representation, wherein the intermediate representation includes a tree that unifies representation of the definition language information and the programming language code, the converter module further checking for syntax errors, and *wherein at least some of the definition language information is represented in the tree without creating new programming language code for the at least some of the definition language information ....*

Grimes and Aho, taken separately or in combination, fail to teach or suggest the above-cited language from each of claims 1, 6, 9, 10, 13, 16, 18, and 20, respectively. For the sake of illustration (and without implying any boundary or limitation on the scope of claim 1), one example of subject matter falling within claim 1 is given on page 24 of the Application:

> In Figure 7, the output module 778 directly manipulates internal compiler structures such as the symbol table 730 and the parse tree 732, creating symbols, adding to the parse-tree, etc.

Another example is given in the Application on page 26:

> While executing concurrently with the compiler, the IDL attribute provider detects (act 852) the occurrence of designated events within the compiler, for example events relat[ing] to the state of compilation .... Examining the state, the IDL attribute provider can wait until the compiler reaches a certain state, and then perform an operation when that state is reached, for example, requesting the compiler to modify the parse tree. The IDL attribute provider then waits for another event.
>
> The IDL attribute provider can perform different operations for different events that might occur within the compiler, and for different parameters transmitted with an IDL attribute. Among these operations are injection of statements or other program elements, possibly employing templates, and modifying or deleting code. Other operations include adding new classes, methods and variables, or modifying existing ones. Modification can include renaming and extending an object or construct.

Grimes describes injecting C++ code in response to attributes generally. [Grimes, pages 2-3.] According to Grimes:

> *When the compiler sees an attribute in your C++ code* it passes the attribute and its arguments to the attribute providers that have been registered on the system. The compiler does this by calling methods on an interface called IAttributeHandler that is implemented by the provider. So that the provider has access to the compiler, it also passes a callback interface pointer of the type ICompiler. If the provider recognizes the attribute, it can then call methods on ICompiler to tell the compiler to *add C++ code to replace the attribute.*

[Grimes, page 3.] This involves replacing an attribute with C++ code when the compiler encounters the attribute. In other words, it involves creating C++ code for the attribute. Replacing an attribute with C++ code when the compiler encounters the attribute (as in Grimes) leads away from the above-cited language of claims 1, 6, 9, 10, 13, 16, and 18, respectively.

In the "Response to Arguments" section of the Action, the Examiner notes that:

> Even if, for the sake of argument only, Applicant were correct in the interpretation of Grimes as adding "programming" code to replace an attribute, the broadest reasonable interpretation of the claim language would still read upon

> the cited prior art. The newly rewritten code would still include, and thus unify through symbol table and a tree, the "definition language information," though perhaps in a different form.

In response, Applicants note that, even under the Examiner's interpretation of Grimes, the use of "newly rewritten code" leads away from the above-cited language of claims 1, 6, 9, 10, 13, 16, and 18, respectively.

Aho does not teach or suggest the above-cited language from claims 1, 6, 9, 10, 13, 16, 18, respectively. Aho generally describes compiler techniques and tools. The parts of Aho specifically cited by the Examiner describe, among other things, compiler organization by "front end" and "back end" [Aho, page 20], symbol tables [Aho, page 11], and parse trees [Aho, pages 6 and 40-48], but do not address working with both programming language and definition language. These parts of Aho additionally do not address the direct manipulation of parse trees and symbol tables in response to IDL attributes.

The Examiner writes:

> Aho demonstrated that it was known at the time of invention to develop compilers with a front end, a converter module, and a back end.... It would have been obvious to one of ordinary skill in the art at the time of invention to implement Grimes' system of C++ code and definition code with a compiler, which would generate executable code as found in Aho's teaching. This implementation would have been obvious because one of ordinary skill would be motivated to provide a mechanism, which would allow source code to produce meaningful executable code.

[Action, page 5.] Elsewhere, the Examiner writes:

> Aho demonstrated that it was known at the time of invention to utilize parse trees and symbol tables.... It would have been obvious to one of ordinary skill in the art at the time of invention to implement Grime's interface definition language / programming language compiler with symbol tables and parse trees as appropriate for compiling such a combination as found in Aho's teaching. This implementation would have been obvious because one of ordinary skill in the art would be motivated to use common and well understood techniques for implementing compilers.

[Action, page 8.] Applicants respectfully disagree. Even if, for the sake of argument, the compiler of Grimes were to be implemented with a symbol table and parse tree as described in Aho (or with any other symbol table and parse tree, as those terms are broadly understood in the art), the symbol table and parse tree would be used for processing C++ code but not for also processing IDL. Grimes makes this clear because it describes (1) processing IDL attributes and

the generated IDL apart from C++ code during compilation and (2) replacing an attribute with C++ code when the attribute is encountered by the compiler. Thus, even for a file with C++ code and IDL attributes, Grimes suggests that any compilation involves only compilation of C++ code. Even if combined, the combination of Grimes and Aho does not teach or suggest the above-cited language of claims 1, 6, 9, 10, 13, 16, and 18, respectively.

For at least these reasons, claims 1, 6, 9, 10, 13, 16, and 18 should be allowable. In view of the foregoing discussion, Applicants will not belabor the merits of the separate patentability of dependent claims 4, 5, 7, 8, 11, 12, 14, 15, 17, 19, and 22-36.

Claim 20

Claim 20 has been rejected under 35 U.S.C. § 103(a) as being unpatentable over Grimes in view of Aho. Grimes and Aho, taken separately or in combination, fail to teach or suggest at least one limitation from claim 20.

Claim 20 recites:

> modifying the programming language compiler to *expose the compiler state to one or more interface definition language attribute providers*; and
> modifying the programming language compiler to allow *manipulation of the symbol table and the parse tree by the one or more interface definition language attribute providers based upon the semantics of the embedded interface definition language information,* wherein the parse tree unifies representation of the interface definition language information and the programming language source code.

Grimes, describes communication between the compiler and the attribute provider [Grimes, pages 2-3.] According to Grimes:

> When the compiler sees an attribute in your C++ code *it passes the attribute and its arguments to the attribute providers that have been registered on the system.* The compiler does this *by calling methods on an interface called IAttributeHandler that is implemented by the provider.* So that the provider has access to the compiler, it also passes a callback interface pointer of the type ICompiler. If the provider recognizes the attribute, *it can then call methods on ICompiler to tell the compiler to add C++ code to replace the attribute.*

[Grimes, pages 2-3.] Thus, according to Grimes, not only does *the compiler call a registered attribute provider* when it "sees" an attribute, but the attribute provider *instructs the compiler to add code* to replace the attribute. Because the attribute provider responds by instructing that code be added, rather than responding with manipulation of any intermediate

representations present in the compiler, the attribute provider described in Grimes *is indifferent to the state of the compiler.* Thus, Grimes does not teach or suggest "modifying the programming language compiler to expose the compiler state to one or more interface definition language attribute providers," as recited in claim 20. Aho also does not teach or suggest the above-cited language of claim 20. For at least these reasons, claim 20 should be allowable.

II. Claim 21 Is Allowable over Grimes

Claim 21 recites:

> embedding by the programming language compiler debugging information in a definition language output file, the definition language output file for subsequent processing by a definition language compiler, whereby the embedded debugging information associates errors raised by the definition language compiler with locations of embedded definition language constructs in the input file to facilitate debugging of the input file.

Claim 21 has been rejected under 35 U.S.C. § 102(a) as being anticipated by Grimes. However, Grimes fails to teach or suggest the above-cited language from claim 21. The Action cites various sections of pages 2 and 3 of Grimes, claiming they map to the language recited in claim 21. Applicants disagree with this characterization of Grimes.

On page 3, Grimes states:

> *The combination of the original and the 'injected' code is compiled to generate an .obj file* and, since this is carried out by the compiler, you do not see any of the generated code. However, in debug builds the compiler will store information about the ATL code *that was generated* in the project's PDB file, so that when you debug attributed code you can step into *the generated code.*

As this passage makes clear, at page 3, Grimes is describing the usage of original and injected *C++ code* to create a .obj file. Because the .obj file is created from C++ code, and not by compilation of IDL information, the stored debug information of page 3 of Grimes would be directed toward the original or injected C++ code itself, and not any IDL information. Indeed, even if IDL information were replaced with injected C++ code, the IDL information would be lost by the time the .obj file is generated.

By contrast, the other cited passages of Grimes discuss "add[ing] attribute information to the .obj file which can be extracted with the Idlgen utility to generate IDL and then run[ing] MIDL to create the type information." [Grimes, page 3.] Applicants note that these passages do not discuss the storing of debug information (and are thus in contrast to the passage of Grimes

addressed in this section above, which does discuss storing of debug information, but not of the type recited in claim 21).

For at least these reasons, Grimes does not teach or suggest each and every limitation of claim 21. Claim 21 should thus be allowable.

III. Claims 1, 4-10, 13-20, 22, 24-25, 27-30, 32-35 Are Allowable over Aho in view of Williams

Claims 1, 4-10, 13-20, 22, 24-25, 27-30, and 32-35 have been rejected under 35 U.S.C. § 103(a) as being unpatentable over Aho in view of Williams. However, Aho and Williams, taken separately or in combination, fail to teach or suggest at least one limitation from each of independent claims 1, 6, 9, 10, 13, 16, 18 and 20 as amended. Thus, the references fail to teach or suggest at least one limitation from each of claims 1, 4-10, 13-20, 22, 24-25, 27-30, and 32-35, and these claims are allowable.

Claim 1 recites "embedded definition language information" as well as "programming language code." In fact, each of claims 6, 9, 10, 13, 16, 18, and 20 recites "definition language constructs" or "definition language information" as well as "programming language constructs" or "programming language code" or "programming language source code" recited in the respective claims. Similarly, for example, the Application, at page 2, lines 2-6 indicates:

> Conventionally, programmers use a definition language to create a specification for an object. This definition language specification defines how the object interacts with other objects. *With reference to this definition language specification, programmers then use a programming language to actually write code for the object.*

The Action admits the recited language of claim 1 is not found in Aho, but claims it is found in Williams. In the rejection of each of the other independent claims, the Action alleges that "[t]he limitations are substantially similar to those of claim 1 and as such are rejected in the same manner." In its rejection of Claim 1, the Action cites to column 2, lines 52-60 of Williams, which reads:

> To allow an object of an arbitrary class to be shared with a client program, interfaces are defined through which an object can be accessed without the need for the client program to have access to the class definitions at compile time. An interface is a named set of logically related function members. *In C++, an interface is an abstract class with no data members and whose virtual functions*

> *are all pure.* Thus, an interface provides a published protocol for two programs to communicate.

Applicants note that the cited passage of Williams references "class definitions" and "a published protocol" but does not anywhere describe a "definition language." In fact, Williams at column 1, lines 52-56 describes class declarations in C++.

> In the C++ language, data encapsulation and inheritance are supported through the use of classes. A class is a user-defined type. A class declaration describes the data members and function members of the class.

As noted above, "definition language," as recited in claim 1, is something other than simple "programming language." Williams does not teach or suggest the "definition language" limitations recited in claims 1, 6, 9, 10, 13, 16, 18 and 20, respectively.

For at least these reasons, Aho and Williams, taken separately or individually, do not teach or suggest at least one limitation of each of claims 1, 6, 9, 10, 13, 16, 18 and 20, respectively. These claims should be allowable. In view of the foregoing discussion, Applicants will not belabor the merits of the separate patentability of dependent claims 4, 5, 7, 8, 14, 15, 17, 19, and 22, 24-25, 27-30, and 32-35.

Request for Information Under 37 C.F.R. § 1.105

The Action, at page 27 and in the Office Action Summary, refers to an "attached requirement for information under 27 CFR 1.105." Applicants have searched for this request, but cannot find it and believe it was not included with the Action due to possible clerical error. If the Office will forward a copy of the Request for Information, Applicants will be happy to reply.

Conclusion

Claims 1 and 4-42 should be allowable. Such action is respectfully requested.

Request for Interview

In view of the preceding amendments and remarks, Applicants believe the application to be allowable. If any issues remain, however, the Examiner is formally requested to contact the undersigned attorney at (503) 226-7391 prior to issuance of the next communication in order to arrange a telephonic interview. This request is being submitted under MPEP § 713.01, which indicates that an interview may be arranged in advance by a written request.

Respectfully submitted,

KLARQUIST SPARKMAN, LLP

By _____
Kyle B. Rinehart
Registration No. 47,027

One World Trade Center, Suite 1600
121 S.W. Salmon Street
Portland, Oregon 97204
Telephone: (503) 226-7391
Facsimile: (503) 228-9446